Beyond Gbps Turbo Decoder on Multi-Core CPUs

Adrien Cassagne*†, Thibaud Tonnellier*, Camille Leroux*, Bertrand Le Gal*, Olivier Aumage† and Denis Barthou†
*IMS Lab, Bordeaux INP, France
†Inria / Labri, Univ. Bordeaux, INP, France

***Abstract*—This paper presents a high-throughput implementation of a portable software turbo decoder. The code is optimized for traditional multi-core CPUs (like x86) and it is based on the Enhanced max-log-MAP turbo decoding variant. The code follows the LTE-Advanced specification. The key of the high performance comes from an inter-frame SIMD strategy combined with a fixed-point representation. Our results show that proposed multi-core CPU implementation of turbo-decoders is a challenging alternative to GPU implementation in terms of throughput and energy efficiency. On a high-end processor, our software turbo-decoder exceeds 1 Gbps information throughput for all rate-1/3 LTE codes with $K < 4096$.**

I. Introduction

Turbo codes [1] are widely used as the channel coding component in digital communication standards, such as the LTE wireless specification [2]. Dedicated hardware architectures of turbo-decoders are usually mapped on custom Silicon in order to reach high energy-efficiency and high-throughput [3]–[6]. In [6] a turbo-decoder ASIC is demonstrated to reach 1.01 Gbps while consuming 0.7 nJ per decoded bit ($K = 6144$ and 6 iterations). However, dedicated hardware turbo-decoders lack flexibility. This will become especially true in the future 5G mobile networks where baseband processing will be virtualized and implemented on centralized cloud platforms [7], [8]. Software implementations will then offer flexibility and short development cycles to the channel decoder design, at the cost of lower throughputs and higher energy consumptions. In such a context, the channel coding functions must be analyzed and optimized to be efficiently mapped on general purpose processors [7], [8]. In the case of turbo coding, some work has been initiated to devise efficient software implementations of turbo decoders, mostly focusing on the LTE standard.

In [9]–[18], turbo decoders were implemented on GPU targets to benefit from their computing power in order to comply with the LTE required throughputs. This was made possible by exploiting the parallelism within the turbo decoding process (*intra-frame* parallelism). An alternative is to process several codewords on distinct computation resources (*inter-frame* parallelism). For instance, in [15], a throughput of 122.8 Mbps was reached for a code dimension $K = 6144$ and 6 decoding iterations on a GPU device. One should notice that this throughput is approximately one decade lower than the dedicated hardware implementation in [6] while the power consumption is one to two decades higher. Indeed, despite the large amount of parallelism in a GPU, it is not obvious to feed every processing unit with data and some non-negligible time is spent in memory accesses. This leads to an inefficient use of the hardware resources and to high energy consumption.

An intermediate solution between dedicated hardware and a GPU is the use of general purpose processors (GPP). Multi-core devices provide high performance computation capabilities while consuming noticeably less power. Thanks to a large set of processor cores it becomes possible to implement computation intensive applications such as channel decoding with a lower energy consumption in comparison to GPU targets. In [16], [19], [20], software turbo decoders are implemented on GPP targets with *intra-frame* parallelism, similar to hardware-oriented strategies. Unlike these works, we propose to investigate the exclusive use of *inter-frame* parallelism. A generic and portable software turbo decoder has been designed and ported on several GPP targets. Experimental results show that inter-frame parallelism allows a more efficient use of CPU resources and our software turbo-decoder outperforms existing implementations in terms of throughput and energy efficiency. Moreover, it exceeds 1 Gbps information throughput on a high-end CPU, making multi-core CPU a compelling alternative to GPU for channel decoding processing in cloud-based Random Access Network (RAN) [7], [8].

The remainder of this paper is organized as follows. Section II presents the turbo code decoding algorithm that was implemented in this work. Section III shows the benefit of inter-frame parallelism for multi-core implementations. Section IV details the optimized implementation of the turbo-decoder. Section V presents experiments and comparison with related works in the field.

II. Overview of the Turbo decoding process

The turbo-decoding process is an iterative process in which two soft input soft output (SISO) decoders exchange extrinsic information. Each SISO decoder uses the channel information and *a priori* extrinsic information to compute *a posteriori* extrinsic information. The *a posteriori* information becomes the *a priori* information for the other SISO decoder and is exchanged via interleaver/deinterleaver.

In turbo-coding, the two component codes are convolutional codes, the associated decoding modules perform the BCJR or forward-backward algorithm [21] which is optimal for the maximum a posteriori (MAP) decoding of convolutional codes. In order to calculate the extrinsic information for a bit, a BCJR SISO decoder first computes the probability that a trellis transition occurred during the encoding process. The branch

metrics associated with states s_i^k and s_j^{k+1} are computed as:

$$\gamma(s_i^k, s_j^{k+1}) = 0.5(L_{sys}^k + L_a^k)u^k + 0.5(L_p^k p^k). \quad (1)$$

Here, L_{sys}^k and L_a^k are the systematic channel LLR and the a-priori LLR for k^{th} trellis section, respectively. In addition, the parity LLRs for the k^{th} trellis step are $L_p^k = L_{p0}^k$ for MAP decoder 0 and $L_p^k = L_{p1}^k$ for MAP decoder 1. We do not need to evaluate the branch metric $\gamma(s^k, s^{k+1})$ for all 16 possible branches, as there are only four different branch metrics: $\gamma_0^k = 0.5(L_{sys}^k + L_a^k + L_p^k)$, $\gamma_1^k = 0.5(L_{sys}^k + L_a^k - L_p^k)$, $-\gamma_0^k$, and $-\gamma_1^k$. After that, the SISO decoder computes forward and backward recursions over the trellis representation of the convolutional code. In this work, we use the Enhanced max-log-MAP algorithm [22], [23]. For each state j of section k of the trellis, the forward (α) and backward (β) metrics are computed as follows:

$$\alpha_j^{k+1} = max_{i \epsilon F}\{\alpha_i^k + \gamma(s_i^k, s_j^{k+1})\} \quad (2)$$

$$\beta_j^k = max_{i \epsilon B}\{\beta_i^{k+1} + \gamma(s_j^k, s_i^{k+1})\}. \quad (3)$$

Then, the extrinsic information for each bit at position k is:

$$\begin{aligned} L_e^k = & max_{\{s_k, s_{k+1}\} \epsilon U^1}\{\alpha_i^k + \beta_j^{k+1} + \gamma(s_i^k, s_j^{k+1})\} \\ & - max_{\{s_k, s_{k+1}\} \epsilon U^{-1}}\{\alpha_i^k + \beta_j^{k+1} + \gamma(s_i^k, s_j^{k+1})\} \\ & - L_{sys}^k - L_a^k, \end{aligned} \quad (4)$$

Finally, L_e is scaled by a fixed factor of 0.75.

III. Parallelism analysis

a) Intra-frame versus inter-frame parallelism: A Turbo decoder is in charge of decoding a large set of frames. Two strategies are then possible to speedup the decoding process. i) *Intra-frame parallelism* : the decoder exploits the parallelism within the turbo-decoding process by executing concurrent tasks during the decoding of one frame. ii) *inter-frame parallelism* : several frames are decoded simultaneously.

In the perspective of a hardware implementation, the intra-frame approach is efficient [24] because the area overhead resulting from parallelization is lower than the speedup. On the contrary, the inter-frame strategy is inefficient, due to the duplication of multiple hardware turbo-decoders. The resulting speedup comes at a high cost in term of area overhead.

In the perspective of a software implementation, the issue is different. The algorithm is executed on a programmable non-modifiable architecture. The degree of freedom lies in the mapping of the different parallelizable tasks on the parallel units of the processor. Modern multi-core processors support Single Program Multiple Data (SPMD) execution. Each core includes Single Instruction Multiple Data (SIMD) units. The objective is then to identify the parallelization strategy suitable for both SIMD and SPMD programming models. In the literature, intra-frame parallelism is often mapped on SIMD units while inter-frame parallelization is usually kept for multi-threaded approaches (SPMD). In [16], [20], multiple trellis-state computations are performed in parallel in the SIMD units. In [9]–[18], [20], the decoded frame is split into sub-blocks

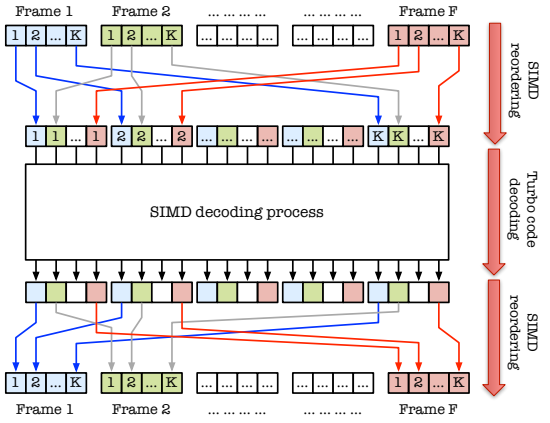

Fig. 1. Frame reordering process before and after the decoding process. Performed 3 times: for systematic, first and second parity information.

that are processed in parallel in the SIMD units. An alternative approach is to process both SISO decoding in parallel but it requires additional computations for synchronization and/or impacts on error-correction performance [24]. However, for all these approaches a part of the computation of the BCJR decoder remains sequential, bounding the speedup beyond the capabilities of SIMD units. Inter-frame parallelism has been proposed in [9], [10], [16], [20]. Multiple codewords are decoded in parallel, it improves the memory access regularity and the usage rate of SIMD units. The speedup is no longer bounded by the sequential parts, all removed, but this comes at the expense of an increase in memory footprint and latency.

In this work, we focus on the inter-frame parallelization and show that the use of this approach allows some register-reuse optimizations that are not possible in the intra-frame strategy.

b) Inter-frame parallelism on multi-core CPUs: The contribution of this work is to propose an efficient mapping of multiple frames on the CPU SIMD units (inter-frame strategy): the decoding of M frames is vectorized. Before the decoding process can be launched, this new approach requires to: (a) buffer a set of M frames and (b) reorder the input LLRs in order to make the SIMDization efficient with memory aligned transactions (see Fig. 1). Similarly, a reversed-reordering step has to be performed at the end of the decoding process. These reordering operations are expensive but they make the complete decoding process very regular and efficient for SIMD parallelization. Moreover, reordering is applied only once, independently of the number of decoding iterations.

Algorithm 1 Standard BCJR implementation

1: **for** all frames **do** ▷ Sequential loop
2: **for** $k = 0;\ k < K;\ k = k + 1$ **do** ▷ Parallel loop
3: $\boldsymbol{\gamma}^k \leftarrow computeGamma(L_{sys}^k, L_p^k, L_e^k)$
4: $\boldsymbol{\alpha}^0 \leftarrow initAlpha()$
5: **for** $k = 1;\ k < K;\ k = k + 1$ **do** ▷ Sequential loop
6: $\boldsymbol{\alpha}^k \leftarrow computeAlpha(\boldsymbol{\alpha}^{k-1}, \boldsymbol{\gamma}^{k-1})$
7: $\boldsymbol{\beta}^{K-1} \leftarrow initBeta()$
8: **for** $k = K - 2;\ k \geq 0;\ k = k - 1$ **do** ▷ Sequential loop
9: $\boldsymbol{\beta}^k \leftarrow computeBeta(\boldsymbol{\beta}^{k+1}, \boldsymbol{\gamma}^k)$
10: **for** $k = 0;\ k < K;\ k = k + 1$ **do** ▷ Parallel loop
11: $L_e^k \leftarrow computeExtrinsic(\boldsymbol{\alpha}^k, \boldsymbol{\beta}^k, \boldsymbol{\gamma}^k)$

Algorithm 2 Loop fusion BCJR implementation

for all frames **do** ▷ Vectorized loop
  $\boldsymbol{\alpha}^0 \leftarrow initAlpha()$
  for $k = 1;\ k < K;\ k = k + 1$ **do** ▷ Sequential loop
    $\boldsymbol{\gamma}^{k-1} \leftarrow computeGamma(L_{sys}^{k-1}, L_p^{k-1}, L_e^{k-1})$
    $\boldsymbol{\alpha}^k \leftarrow computeAlpha(\boldsymbol{\alpha}^{k-1}, \boldsymbol{\gamma}^{k-1})$
  $\boldsymbol{\gamma}^{K-1} \leftarrow computeGamma(L_{sys}^{K-1}, L_p^{K-1}, L_e^{K-1})$
  $\boldsymbol{\beta}^{K-1} \leftarrow initBeta()$
  $L_e^{K-1} \leftarrow computeExtrinsic(\boldsymbol{\alpha}^{K-1}, \boldsymbol{\beta}^{K-1}, \boldsymbol{\gamma}^{K-1})$
  for $k = K - 2;\ k \geq 0;\ k = k - 1$ **do** ▷ Sequential loop
    $\boldsymbol{\beta}^k \leftarrow computeBeta(\boldsymbol{\beta}^{k+1}, \boldsymbol{\gamma}^k)$
    $L_e^k \leftarrow computeExtrinsic(\boldsymbol{\alpha}^k, \boldsymbol{\beta}^k, \boldsymbol{\gamma}^k)$

In the proposed implementation, the inter-frame parallelism is used to fill the SIMD units of the CPU cores. Algorithm 1 illustrates the traditional implementation of the BCJR (used for the *intra-frame* vectorization). The inter-frame strategy makes the outer loop on the frame parallel (through vectors). This means all computations inside this loop operate on SIMD vectors instead of scalars, and the inner loops can be turned into sequential loops on SIMD vectors. This gives the opportunity for memory optimizations, through loop fusion. The initial 4 inner loops are merged into 2 loops. Algorithm 2 presents this loop fusion optimization. This makes possible the scalar promotion of β_j (no longer an array), since it can be directly reused from the CPU registers. In this version, the SIMD are always stressed.

On a multicore processor, each core decodes M frames using its own SIMD unit and T threads are activated, a total of $M \times T$ frames are therefore decoded simultaneously with the inter-frame strategy. Theoretically, this SPMD parallelization strategy provide an acceleration up to a factor T, with T cores. Large memory footprint, exceeding L3 cache capacity may reduce the effective speedup, as shown in Section V.

IV. Implementation of the decoder

The presented decoder implementation is available in the AFF3CT[1] software [25]. The use of `C++` templates associated to our generic SIMD library enables the same source code to be compiled using different formats (32-bit `float`, 16-bit `short`, and 8-bit `char`) and different SIMD instructions (SSE, AVX and NEON), providing possible trade-offs between SIMDization, throughput and error-correction performance.

a) Fixed-point representation: Nowadays on x86 CPUs, there are large SIMD registers: SSE/NEON are 128 bits wide and AVX are 256 bits wide. The number of elements that can be vectorized depends on the SIMD length and on the data format: $n_{elem} = sizeof(SIMD)/sizeof(data)$. So, the key for a large parallelism is to work on short data.

As there is no floating-point support for 16-bit and 8-bit data, a fixed-point representation is used. The AWGN channel soft information is quantized as follows: $y_{s,v}^k = \Psi(2^v.y^k \pm 0.5)$, with y^k the current floating-point value from the channel, s the number of bit of the quantized number, including v bits for the fractional part and the saturation function $\Psi(x) = min(max(x, -2^{s-1}+1), 2^{s-1}-1)$. In the experiments (cf. Fig. 2) $Q_{s,v}$ denotes this channel quantization.

During the turbo-decoding process, the extrinsic values grow at each iteration. It is then necessary for internal LLRs to have a larger dynamic than the channel information. Depending on data format, 16-bit or 8-bit, the quantization used in the decoder is $Q_{16,3}$ or $Q_{8,2}$, respectively.

b) Memory allocations: The systematic information L_{sys_N}/L_{sys_I} and the parity information L_{p_N}/L_{p_I} are stored in the natural domain N as well as in the interleaved domain I. Two extrinsic vectors are also stored: L_{e_N} in N and L_{e_I} in I. Inside the BCJR decoding and per trellis section, two γ_i and eight α_j metrics are stored. Thanks to the loop fusion optimization, the eight β_j metrics are not stored in the global memory. In the proposed implementation $i \in \{0,1\}$ and $j \in \{0,1,2,3,4,5,6,7\}$. Notice that all those previously-mentioned vectors are K-bit wide and are duplicated $M \times T$ times because of the inter-frame strategy. The memory footprint in bytes is approximatively equal to: $16 \times K \times sizeof(data) \times M \times T$. The interleaving and deinterleaving lookup tables have been neglected in this model.

c) Forward trellis traversal: The objective is to reduce the number of loads/stores, performing the arithmetic computations (`add` and `max`) inside registers. The max-log-MAP algorithm only stresses the integer pipeline of the CPU. This kind of operations takes only one cycle to execute when the latency is also very small (1 cycle too). In contrast, a load/store can take a larger number of cycles depending on where the current value is loaded/stored in the memory hierarchy. Using data directly from the registers is cost-free but loading/storing it from the L1/L2/L3 cache can take up to 30 cycles (at worst).

Per trellis section k, the two γ_i^k metrics are computed from the systematic and the parity information. These two γ_i^k are directly reused to compute the eight α_j^k metrics. Depending on the number of bits available, the trellis traversal requires to normalize the α_j^k because of the accumulations along the multiple sections. In 8-bit format, the α_j^k metrics are normalized for each section: the first α_0^k value is subtracted to all the α_j^k (including α_0^k itself). In the 16-bit decoder, the normalization is only applied every eight steps (like in [16]), since there are enough bits to accumulate eight values. We have observed in experiments that there is no performance degradation due to the normalization process. At the end of a trellis section k the two γ_i^k and the eight normalized α_j^k are stored in memory. In the next trellis section $(k+1)$ the eight previous α_j^k are not loaded from memory but they are directly reused from registers to compute the α_j^{k+1} values.

d) Backward trellis traversal: Per trellis section k, the two γ_i^k metrics are loaded from the memory. These two metrics are then used to compute, on the fly, the eight β_j^k metrics (whenever needed the β_j^k metrics have been normalized like for the α_j^k metrics). After that, the α_j^k metrics are loaded from the memory. The α_j^k, β_j^k and γ_i^k metrics are used to determine the *a posteriori* and the extrinsic LLRs. In the next trellis section $(k-1)$ the previous β_j^k metrics are directly reused

[1] AFF3CT is an Open-source software (MIT license) for fast forward error correction simulations, see `http://aff3ct.github.io`

TABLE I
SPECIFICATIONS OF THE TARGET PROCESSORS.

CPU	**P1** : Xeon E5-2650	**P2**: Core i7-4960HQ	**P3**: Xeon E5-2680v3
Intel Arch.	*Ivy Bridge* Q1'12	*Haswell* Q4'13	*Haswell* Q3'14
Cores/Freq.	8 cores, 2–2.8 GHz	4 cores, 2.6–3.8 GHz	12 cores, 2.5–3.3 GHz
LLC	20MB L3	6MB L3	30MB L3
TDP	95 W	47 W	120 W

from registers in order to compute the next β_j^{k-1} values. The β_j^k metrics are then never stored in the memory.

V. EXPERIMENTS AND RESULTS

The experiments have been conducted on three different x86-based processors detailed in Table I. A mid-range processor (P2) is used for comparison with similar CPU targets in the literature [16], [19], [20] while the two high-end processors (P1 and P3) are used for comparison with GPU-based turbo-decoder implementations. Indeed, P1 and P3 have a number of cores that is similar to the number of *Streaming Multiprocessors* (SM) inside a GPU. Moreover, the code has been compiled on Linux (Ubuntu 14.04 LTS) with the GNU compiler (version 4.8) and with the `-Ofast -funroll-loops -msse4.1/-mavx2` flags.

a) BER/FER performance: Fig. 2 shows the decoding performance of the proposed software turbo-decoder for the $K = 6144$ rate-1/3, LTE-specified turbo-code. The decoding performance of a floating-point decoder is provided as a reference. Unlike [16], the proposed 16-bit implementation does not degrade the decoding performance. The 8-bit version of our decoder shows a 0.15dB degradation. The limited dynamic of 8-bit format together with early saturation inside the decoder are responsible for this small performance loss.

b) Throughput performance: Fig. 3 shows the evolution of the information throughput depending on the code dimension K. This experiment was conducted on P2 and P3 (both have *Haswell* architectures). The throughput tends to increase linearly with the number of cores (up to 24 cores) except in AVX mode where a performance drop can be observed when $K > 4096$. The reason is that the AVX instructions

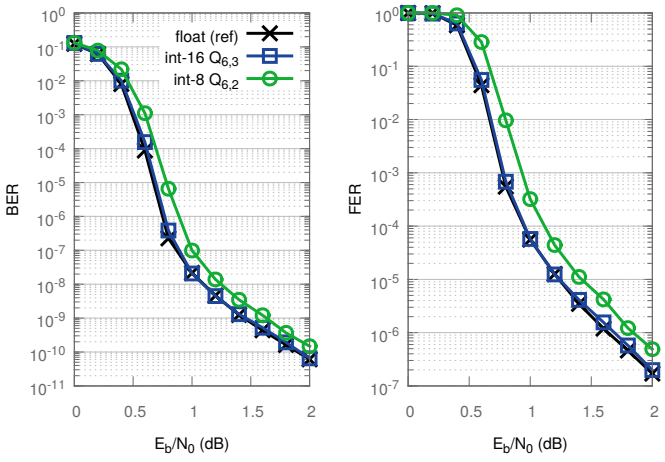

Fig. 2. Bit Error Rate (BER) and Frame Error Rate (FER) of the decoder for K = 6144 (6 iters) and $R = 1/3$. Enhanced max-log-MAP algorithm (scaling factor = 0.75). BPSK modulation and AWGN channel were used.

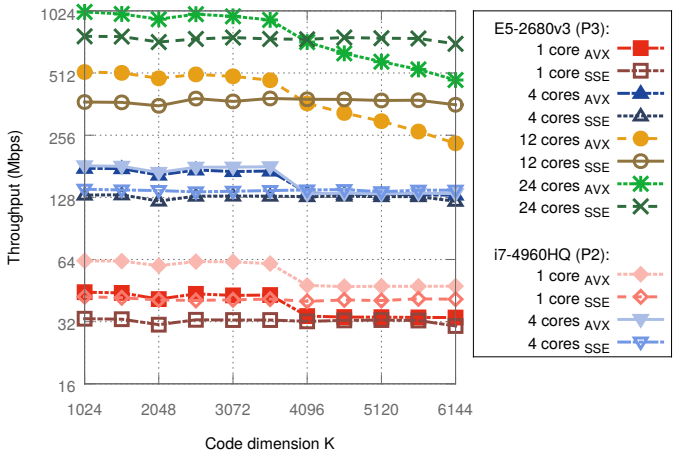

Fig. 3. Information throughput depending on K for various number of cores and SIMD instruction types. 6 iterations, 8-bit fixed-point.

use vectors $2\times$ wider than those used by SSE instructions and the inter-frame strategy loads twice the number of frames to fill these vectors. Thus, for $K > 4096$, in AVX, the memory footprint exceeds the L3 cache optimal occupancy and the performance is driven by the RAM bandwidth. Then, as K increases the number of RAM accesses increases and there is not enough memory bandwidth to feed all the cores. This explains the decreasing throughput for $K > 4096$, in AVX mode. Nonetheless, on P3 target, the throughput exceeds 1Gbps for all codes with $K < 4096$.

Fig. 4 shows the energy consumed by the processor to decode one information bit (E_d) of the codes using SSE and AVX instructions, on the P2 CPU target. For small codewords ($K = 1024$) it is more energy efficient to resort to AVX. But this is not so clear on larger codewords ($K = 6144$) since with 3/4 cores, the code using SSE outperforms the AVX one.

Table II shows a performance comparison with related works[2]. The variety of CPU/GPU targets and algorithmic parameters allows to show some global emerging trends. When comparing to similar CPU targets [16], [20], the proposed im-

[2]To be as fair as possible with the other works, we assume that the *Intel Turbo Boost* (ITB) technology was disabled on their CPUs. For our experiments, the ITB technology was on and the real frequency is picked up. Moreover, for GPU works there is an asterisk when it is unclear if the CPU/GPU data transfer times have been taken into account.

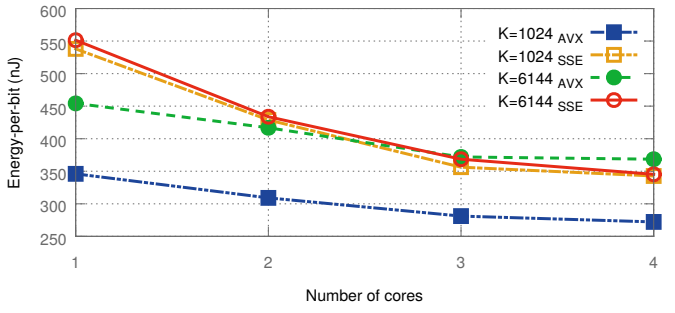

Fig. 4. *Energy-per-bit* (E_d) depending on the number of cores and the instruction types. 6 iterations, 8-bit fixed-point. The throughput and power measurements were conducted on P2 with the *Intel Power Gadget* tool.

TABLE II

PERFORMANCE COMPARISON WITH THE OTHER WORKS. ALL THE REPORTED METRICS ARE NORMALIZED TO ONE ITERATION. $NThr. = (Thr. \times Iters)/(Freq. \times Cores)$, $TNDC = (Thr. \times Iters)/(Cores \times Freq. \times SIMD)$, $E_d = [TDP/(Thr. \times Iters)] \times 10^3$. ON CPUS, ONLY SSE INSTRUCTIONS ARE CONSIDERED.

	Work	Year	Hardware and decoder parameters											Decoding performances				Metrics		
			Platform	Arch.	TDP Watts	Cores or SM	Freq. GHz	Algorithm	Pre. bit	SIMD length	Inter level	K	Iters	BER at 0.7 dB	FER at 0.7 dB	Lat. μs	Thr. Mbps	NThr. Mbps	TNDC	E_d nJ
GPU-based	[9]	2010	Tesla C1060	*Tesla*	200	15	1.30	ML-MAP	32	16	100	6144	5	1e-04	–	76800	8.0	2.1	0.135	5000
	[10]	2011	GTX 470	*Fermi*	215	14	1.22	ML-MAP	32	32	100	6144	5	4e-05	–	20827	29.5	8.6	0.270	1458
	[11]	2012	Tesla C2050	*Fermi*	247	14	1.15	L-MAP	32	32	32	11918	5	–	–	108965	3.5	1.1	0.035	14114
	[12]	2012	9800 GX2	*Tesla*	197	16	1.50	ML-MAP	32	16	1	6144	5	1e-02	–	3072	2.0	0.4	0.025	19700
	[13]	2013	GTX 550 Ti	*Fermi*	116	6	1.80	EML-MAP	32	32	1	6144	6	1e-02	–	72*	85.3	47.4	1.482	227
	[14]	2013	GTX 580	*Fermi*	244	16	1.54	ML-MAP	32	32	1	6144	6	3e-04	–	1660	3.7	0.9	0.030	10090
	[15]	2013	GTX 480	*Fermi*	250	15	1.40	EML-MAP	32	32	1	6144	6	–	–	50*	122.8	35.1	1.098	339
	[16]	2013	GTX 680	*Kepler*	195	8	1.01	EML-MAP	32	192	16	6144	6	–	1e-02	2657	37.0	27.5	0.144	878
	[17]	2014	Tesla K20c	*Kepler*	225	13	0.71	ML-MAP	32	192	1	6144	5	1e-04	–	1097	5.6	3.0	0.015	8036
	[18]	2014	GTX 580	*Fermi*	244	16	1.54	BR-SOVA	8	32	4	6144	5	2e-02	–	192*	127.8	25.9	0.810	382
CPU-based	[19]	2011	i7-960	*Nehalem*	130	1	3.20	ML-MAP	16	8	1	1008	8	3e-03	7e-02	138	7.3	18.3	2.280	2226
	[20]	2012	X5670	*Westmere*	95	6	2.93	EML-MAP	8	16	6	5824	3	6e-02	–	157	222.6	38.0	2.373	142
	[16]	2013	i7-3770K	*Ivy Bridge*	77	4	3.50	EML-MAP	16	8	4	6144	6	–	1e-01	323	76.2	32.7	4.080	168
	this work	2016	E5-2650	*Ivy Bridge*	95	8	2.50	EML-MAP	16	8	64	6144	6	6e-06	6e-03	3665	107.3	32.2	4.014	148
			i7-4960HQ	*Haswell*	47	4	3.20				32					2212	88.9	41.7	5.208	88
			2×E5-2680v3	*Haswell*	240	24	2.50				192					2657	443.7	44.4	5.544	90
			E5-2650	*Ivy Bridge*	95	8	2.50		8	16	128			8e-05	5e-02	3492	225.2	67.6	4.224	70
			i7-4960HQ	*Haswell*	47	4	3.20				64					2837	138.6	65.0	4.062	57
			2×E5-2680v3	*Haswell*	240	24	2.50				384					3293	716.4	71.6	4.476	56

plementation reaches similar or higher throughput (from 88.9 Mbps to 138.6 Mbps on P2 target) at the price of an increased latency (from 2212 μs to 2837 μs) and memory footprint. The proposed high-end CPU processor (P3) implementation outperforms all GPU-based works in terms of throughput (from 443.7 Mbps to 716.4 Mbps) while consuming noticeably less power (from 56 nJ to 90 nJ for each iteration). This leads to the conclusion that high-end multi-core CPUs is a more energy-efficient solution than GPUs while ensuring similar or higher throughputs. Considering this, high-end multi-core CPU appear as an alternative to GPU in future channel coding functions in cloud-based RAN.

VI. CONCLUSION

Future communication standards will make use of cloud-based RAN. In such a context, channel decoding processing will be mapped on programmable targets. In this work, we investigate the use of inter-frame parallelism for optimized software implementation of turbo decoders. Our results show that CPUs are competitive solutions in terms of throughput and energy consumption. The proposed software decoder exceeds 1 Gbps on a high-end CPU.

ACKNOWLEDGMENT

This work was supported by a grant overseen by the French National Research Agency (ANR), ANR-15-CE25-0006-01.

REFERENCES

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes." in *IEEE ICC*, 1993.

[2] ETSI, "3GPP - TS 136.212 - Multiplexing and channel coding (R. 11)."

[3] C. Benkeser, A. Burg, T. Cupaiuolo, and Q. Huang, "Design and optimization of an HSDPA turbo decoder ASIC," *IEEE JSSC*, vol. 44, no. 1, pp. 98–106, 2009.

[4] C. Studer, C. Benkeser, S. Belfanti, and Q. Huang, "Design and implementation of a parallel turbo-decoder ASIC for 3GPP-LTE," *IEEE JSSC*, vol. 46, no. 1, pp. 8–17, 2011.

[5] Y. Sun and J. R. Cavallaro, "Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder," *Integration JVLSI*, vol. 44, no. 4, pp. 305–315, 2011.

[6] S. Belfanti *et al.*, "A 1Gbps LTE-advanced turbo-decoder ASIC in 65nm CMOS," in *IEEE VLSIC*, 2013.

[7] H. Paul, D. Wubben, and P. Rost, "Implementation and analysis of forward error correction decoding for cloud-RAN systems," in *IEEE ICCW*, 2015.

[8] D. Wubben *et al.*, "Benefits and impact of cloud computing on 5g signal processing: Flexible centralization through cloud-ran," *IEEE SPM*, vol. 31, no. 6, pp. 35–44, 2014.

[9] M. Wu, Y. Sun, and J. R. Cavallaro, "Implementation of a 3GPP LTE turbo decoder accelerator on GPU," in *IEEE SiPS*, 2010.

[10] M. Wu, Y. Sun, G. Wang, and J. R. Cavallaro, "Implementation of a high throughput 3GPP turbo decoder on GPU," *Springer JSPS*, vol. 65, no. 2, pp. 171–183, 2011.

[11] S. Chinnici and P. Spallaccini, "Fast simulation of turbo codes on GPUs," in *IEEE ISTC*, 2012, pp. 61–65.

[12] D. Yoge and N. Chandrachoodan, "GPU implementation of a programmable turbo decoder for software defined radio applications," in *IEEE VLSID*, 2012.

[13] C. Liu, Z. Bie, C. Chen, and X. Jiao, "A parallel LTE turbo decoder on GPU," in *IEEE ICCT*, 2013.

[14] X. Chen, J. Zhu, Z. Wen, Y. Wang, and H. Yang, "BER guaranteed optimization and implementation of parallel turbo decoding on GPU," in *IEEE ICST*, 2013.

[15] J. Xianjun, C. Canfeng, P. Jaaskelainen, V. Guzma, and H. Berg, "A 122mb/s turbo decoder using a mid-range GPU," in *IEEE IWCMC*, 2013.

[16] M. Wu, G. Wang, B. Yin, C. Studer, and J. R. Cavallaro, "HSPA+/LTE-A turbo decoder on GPU and multicore CPU," in *IEEE ACSSC*, 2013.

[17] Y. Zhang *et al.*, "The acceleration of turbo decoder on the newest GPGPU of kepler architecture," in *IEEE ISCIT*, 2014.

[18] R. Li, Y. Dou, J. Xu, X. Niu, and S. Ni, "An efficient parallel SOVA-based turbo decoder for software defined radio on GPU," *IEICE Trans. Fundamentals*, vol. 97, no. 5, pp. 1027–1036, 2014.

[19] L. Huang *et al.*, "A high speed turbo decoder implementation for CPU-based SDR system," in *IEEE IET ICCTA*, 2011.

[20] S. Zhang, R. Qian, T. Peng, R. Duan, and K. Chen, "High throughput turbo decoder design for GPP platform," in *IEEE ICST*, 2012.

[21] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate (corresp.)," *IEEE TIT*, vol. 20, no. 2, pp. 284–287, 1974.

[22] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *IEEE ICC*, 1995.

[23] J. Vogt and A. Finger, "Improving the max-log-MAP turbo decoder," *Electronics Letters*, vol. 36, no. 23, pp. 1937–1939, 2000.

[24] O. Muller, A. Baghdadi, and M. Jezequel, "From parallelism levels to a multi-asip architecture for turbo decoding," *IEEE TVLSIS*, vol. 17, no. 1, pp. 92–102, 2009.

[25] AFF3CT, "AFF3CT: The first software release," 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.55668